

Ingenieria II - Apuntes

20 de marzo de 2020

Índice

1. Introduccion	3
1.1. Model Checking	3
2. Programas y sistemas concurrentes	5
I Labeled Transition System (LTS)	6
3. Definiciones y propiedades	6
3.1. Ejecuciones y trazas	6
3.2. Composición paralela	7
4. Finite State Process (FSP)	8
5. Verificación de propiedades	14
5.1. Propiedades de safety	14
5.1.1. Property	14
5.2. Propiedades de Liveness	15
5.2.1. Progress	15
5.3. Bisimulación	16
5.3.1. El juego de la bisimilaridad	17
II Lógica Temporal Lineal	18
6. Estructuras de Kripke	18
7. Lógica temporal lineal (LTL)	20
7.1. Semántica	20

8. Verificación de propiedades	21
8.1. Autómata de buchi	21
8.2. Método de verificación	21
 III Lógica Computacional Árborea (CTL)	 24
9. Sintaxis y semantica	24
9.1. Semántica	24
9.1.1. Equivalencias	25
10.Verificación explícita de propiedades	26
10.1. Fórmulas como conjuntos característicos	26
10.1.1. Algoritmo de verificación explícito:	27
11.Verificación implícita de propiedades	29
11.1. Árboles binarios de decision	29
11.1.1. Construcción de un ROBDD	30
11.2. Cálculo de EX ψ	34
11.2.1. Conversión de un Kripke a fórmulas booleanas	34
11.2.2. Eliminación del existencial	35
11.3. Algoritmo de punto fijo	36
11.3.1. Los puntos fijos de un reticulado	37
11.3.2. Operadores CTL definidos recursivamente	38
11.3.3. Operadores CTL como puntos fijos	38

1. Introduccion

Sistemas reactivos: Sistemas que reaccionan a estímulos provenientes de su entorno. Nos referimos a los estímulos y reacciones como **eventos** o **acciones**.

1.1. Model Checking

El **model checking** es una técnica para verificar sistemas concurrentes con estados finitos. El procedimiento, normalmente usa una búsqueda exhaustiva sobre el espacio de los estados del sistema para determinar si una especificación es correcta o no. Si se tienen los recursos suficientes (de hardware), entonces siempre termina con una respuesta que puede ser **si** ó **no**. Además, puede ser implementada con algoritmos razonablemente eficientes que puede ser corridos en máquinas promedio.

Aplicar la comprobación a un diseño consiste en varias tareas:

Modelado: Convertir el diseño en un formalismo aceptado por la herramienta de comprobación (En nuestro caso el FSP). En muchos casos, esto es solo una compilación. En otros, debido a limitaciones de tiempo y memoria, puede requerir el uso de ciertas abstracciones para eliminar detalles irrelevantes al problema que tratamos de resolver.

Modelo: Representación simplificada del mundo real que incluye solo aquellos aspectos del sistema que son relevantes al problema.

Proceso: Ejecución de un programa secuencial de un programa. El estado de un proceso, en cualquier momento del tiempo, consiste en los valores de variables explícitas (declaradas por el programador) y variables implícitas (program counter, registros de direcciones, etc).

Los modelos que vamos a usar ignora estos detalles de implementación y considera que un cambio de estado es producido por acciones atómicas e indivisibles. Cada acción causa una transición desde el estado actual hacia el siguiente. El orden en el que ocurren estas acciones se determina mediante un **grafo de transiciones**.

Especificación: Es necesario declarar las propiedades que debe satisfacer el diseño. Esto se hace usando un formalismo lógico. En sistemas de hardware y software, es común usar **lógica temporal** con la cual se puede describir cómo un sistema evoluciona con el tiempo.

Uno de las cuestiones más importantes de la especificación es la **completitud**. El model checking es una herramienta que nos permite determinar si un modelo del diseño satisface una especificación dada, pero es imposible determinar si ésta cumple con todas las propiedades que el sistema debería satisfacer.

Verificación: Idealmente, es automática. Sin embargo, un humano debe realizar un análisis de los resultados de la verificación. En caso de un resultado negativo, el usuario es provisto de una traza de error (**error trace**) que puede ser usado como contra ejemplo para la propiedad fallida.

Una corroboración de una propiedad puede fallar por varios motivos:

- **Diseño incorrecto:** El sistema diseñado no refleja realmente el comportamiento deseado, por lo que debe ser modificado.
- **Modelo incorrecto:** El diseño es correcto, sin embargo, al escribirlo en el lenguaje de la herramienta, no se implementó correctamente.
- **Especificación incorrecta (*Falso negativo*):** La propiedad que falló, en realidad, fue mal especificada y es correcto que no funcione para esos casos. Se debe revisar la especificación para que describa correctamente el comportamiento deseado.

2. Programas y sistemas concurrentes

La mayoría de los sistemas complejos y tareas que ocurren en el mundo físico pueden ser divididos en conjuntos de actividades más simples. Las mismas, no ocurren secuencialmente, sino que se superponen en el tiempo y se realizan de manera concurrente. Lo mismo sucede con muchos de los programas ejecutados por una computadora.

La ejecución de un programa (o subprograma) se denomina **proceso** y la ejecución de un programa concurrente consiste en múltiples procesos que se comunican entre sí para reflejar el comportamiento deseado según los eventos que le llegan.

Estructurar un programa de esta manera tiene varias ventajas:

- Simplifica el modelado de sistemas reactivos.
- La concurrencia puede ser usada para mejorar el tiempo de respuesta de una aplicación derivando tareas que consumen demasiado tiempo a otros procesos.
- La comunicación entre procesos y la realización de tareas en paralelo, mejora el throughput de operaciones realizadas.
- Se puede aprovechar mejor, el hardware diseñado para realizar multi-tasking.

Parte I

Labeled Transition System (LTS)

3. Definiciones y propiedades

Son autómatas finito con transiciones etiquetadas que representan un programa. Para cada estado, habrá un conjunto de etiquetas que nos indicará las acciones aceptadas por ese programa cuando se encuentra en ese estado y el/los estados a los que pasa cuando se realiza alguna de esas acciones.

Además, agregamos τ al conjunto de etiquetas para modelar el pasaje de estados por acciones internas del programa (no visibles desde el entorno). Es decir, para representar operaciones que el programa realiza automáticamente cuando llega a un estado, sin la necesidad de ningún estímulo externo extra.

DEFINICIÓN: LTS (Labeled Transition System)

Sea Q el universo de estados, Σ el universo de acciones observables y $\Sigma_\tau = \Sigma \cup \{\tau\}$. Un **LTS** es una tupla $P = (Q', \Sigma', \Delta, q_0)$ donde $Q' \subseteq Q$ es un conjunto finito, $\Sigma' \subseteq \Sigma_\tau$ es un conjunto de etiquetas, $\Delta \subseteq (Q \times \Sigma \times Q)$ es un conjunto de transiciones etiquetadas y s_0 es el estado inicial. Definimos el **alfabeto de comunicación** de P como $\alpha P = \Sigma' \setminus \{\tau\}$.

3.1. Ejecuciones y trazas

Una ejecución es la secuencia de todos los estados por los que pasó el automatá junto con las acciones que provocaron esas transiciones y una traza es la secuencia de acciones que fueron realizadas durante la ejecución del programa.

DEFINICIÓN: Ejecución y Traza

Una **ejecución** de un LTS $P = (Q', \Sigma', \Delta, s_0)$ es una secuencia de estados y etiquetas $s_0, l_0, s_1, l_1, \dots$ tal que para cada $i \geq 0$ existe una transición del estado s_i al s_{i+1} con etiqueta l_i . Una secuencia l_0, l_1, l_2, \dots es una **traza** de P si es una proyección sobre αP de una ejecución de P .

DEFINICIÓN: Transitar

(Meh, formalismo de las transiciones)

Dado un LTS $P = (Q', \Sigma', \Delta, s_0)$, decimos que $P \xrightarrow{l} P'$ (transita con la acción $l \in \Sigma'$ a un LTS P') si $P' = (Q', \Sigma', \Delta, s')$ y $(s, l, s') \in \Delta$.

3.2. Composición paralela

La composición paralela entre dos autómatas M y N nos devuelve el autómata $M||N$ tal que:

- $Q_{M||N} \subseteq Q_M \times Q_N$, los estados de $M||N$ pertenece al producto cartesiano de los estados de los autómatas originales.
- Las etiquetas del nuevo autómata es la unión de las etiquetas de los originales.
- Si m_0 y n_0 son los estados iniciales de M y N , respectivamente, entonces el estado inicial $M||N$ es la tupla (m_0, n_0) .
- Si agarramos el estado (s, t) de $M||N$, entonces pasa alguno de los siguientes:
 - Si, en M , hay una transición desde el estado s a uno s' por una etiqueta l , pero no hay ninguna transición, en N , desde t que use esta etiqueta entonces (s, t) transiciona a (s', t) por l .
 - Analogamente, si esta transición existe en N y no en M , entonces (s, t) transiciona a (s, t') por l .
 - Si existe una transición en ambos autómatas, entonces (s, t) transiciona (s', t') por l .

DEFINICIÓN: Composición paralela

Sean los LTS $M = (Q_M, \Sigma_M, \Delta_M, m_0)$ y $N = (Q_N, \Sigma_N, \Delta_N, n_0)$. La **composición paralela** $M||N$ de M y N es el LTS tal que:

$$M||N = (Q_M \times Q_N, \Sigma_M \cup \Sigma_N, \Delta, (m_0, n_0))$$

donde Δ es la relación más chica que satisface las siguientes reglas:

$$\frac{(s, l, s') \in \Delta_M}{((s, t), l, (s', t)) \in \Delta} \quad l \notin \alpha N \quad \frac{(t, l, t') \in \Delta_N}{((s, t), l, (s, t')) \in \Delta} \quad l \notin \alpha M$$

$$\frac{(s, l, s') \in \Delta_M \quad (t, l, t') \in \Delta_N}{((s, t), l, (s', t')) \in \Delta} \quad l \in \alpha M \cap \alpha N$$

Propiedades: La composición paralela es conmutativa y asociativa.

Acciones compartidas: Si dos procesos en una composición tienen acciones en común, se dice que éstas son **compartidas**. En nuestro modelo, usaremos las acciones compartidas para simular la interacción entre procesos.

Mientras que las acciones no compartidas pueden ocurrir en cualquier momento y orden. Una acción compartida debe ser ejecutada al mismo tiempo por todos los procesos que comparten esa acción.

4. Finite State Process (FSP)

Es el lenguaje que usamos para describir LTS's. Nos proporciona procesos y operadores básicos que nos permiten definir procesos más complejos.

La traducción de un LTS a un FSP se puede definir como una función $lts : FSP \rightarrow LTS$ que toma una expresión de FSP y la convierte en un LTS.

- Los nombres de los procesos que definamos deben empezar con mayúscula.
- Los nombres de las acciones deben empezar con minúscula.
- Las definiciones de todos los procesos terminan en un punto.

STOP Es el proceso más simple. No hace nada.

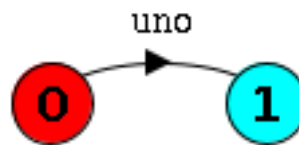
DEADLOCK = STOP.



Action prefix Si x es una acción y P un proceso entonces $(x \rightarrow P)$ describe un proceso que inicialmente interactúa a través de la acción x y luego se comporta como P .

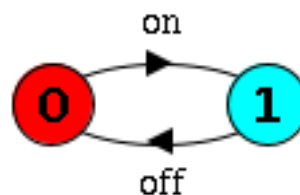
Todas las definiciones de procesos que creemos deben finalizar con un proceso (puede ser cualquier subproceso local, el mismo proceso o alguno de los reservados como **STOP** y **ERROR**)

UnicaVez = (uno \rightarrow STOP).



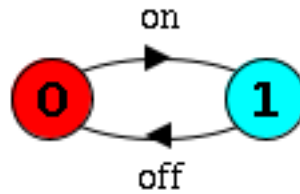
Recursión Sirve para modelar repetición de comportamiento

SWITCH = (on \rightarrow off \rightarrow SWITCH)



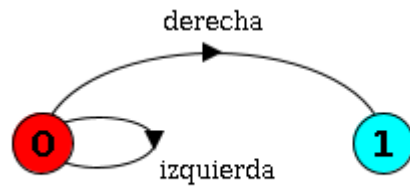
Subprocesos Cuando se debe separar un proceso en subprocesos, se lo puede hacer separándolos con una coma. Cada subproceso declarado solo es válido localmente, es decir, pertenecen a la definición del proceso original y no pueden ser usados por otros.

$\text{SWITCH} = \text{OFF},$
 $\text{OFF} = (\text{on} \rightarrow \text{ON}),$
 $\text{ON} = (\text{off} \rightarrow \text{OFF}).$



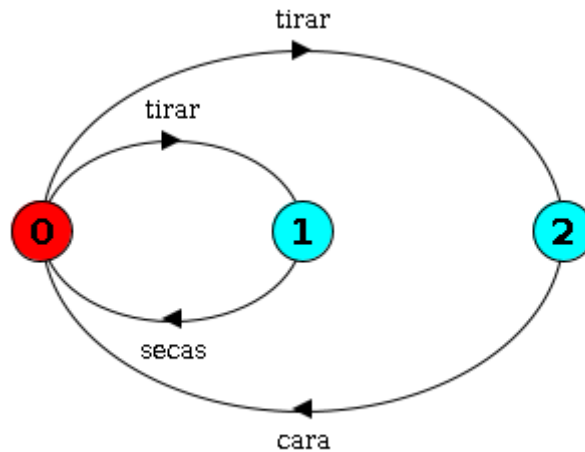
Alternativa Si x e y son acciones entonces $(x \rightarrow P \mid y \rightarrow Q)$ describe un proceso que inicialmente es capaz de interactuar a través de las acciones x ó y .

$\text{GIRO} = (\text{izquierda} \rightarrow \text{GIRO} \mid \text{derecha} \rightarrow \text{STOP}).$



Elección no determinística El proceso $(x \rightarrow P \mid x \rightarrow Q)$ describe una elección no determinística entre P ó Q .

$\text{MONEDA} = (\text{tirar} \rightarrow \text{CARA} \mid \text{tirar} \rightarrow \text{SECAS}),$
 $\text{CARA} = (\text{cara} \rightarrow \text{MONEDA}),$
 $\text{SECAS} = (\text{secas} \rightarrow \text{MONEDA}).$



Procesos y acciones indexados: Nos permiten modelar procesos y acciones que pueden tomar múltiples valores. Los índices siempre tienen un rango finito.

$\text{BUFF} = (\text{in}[i:0..3] \rightarrow \text{out}[i] \rightarrow \text{BUFF})$

Declaración de constantes y rangos: Definimos constantes que pueden ser usadas en cualquier lugar del modelo. Para valores únicos usamos **CONST** y para rangos **RANGE**.

```
CONST N = 1
RANGE T = 0..N
RANGE R = 0..2*N
```

```
SUM = (in[a: T] [b:T] -> TOTAL[a + b]),
TOTAL[s:R] = (out[s] -> SUM)
```

Parámetros de procesos Se pueden describir procesos de manera general y pasarle como parámetro una constante inicializada en algún valor para modelar una instancia particular. El siguiente FSP genera un LTS para un buffer de tres elementos.

```
BUFF(N=3) = (in[i: 0..N] -> out[i] -> BUFF).
```

Guardas Se pueden definir acciones de manera condicional, dependiendo del estado actual de una máquina. Usamos guardas booleanas **when(B) x ->P** para indicar que una acción particular **x** puede ser elegida si solo si la guarda **B** es satisfecha.

```
COUNT(N=3) = COUNT[0],
COUNT[i:0..N] = (
    when(i < N) inc -> COUNT[i+1] |
    when(i > 0) dec -> COUNT[i-1]
).
```

Alfabetos En LTS tenemos un operador de extensión de alfabeto que extiende el alfabeto implícito de un LTS con las etiquetas deseadas:

```
WRITER = (write[1] -> write[3] -> WRITER) + {writer[0..3]}.
```

Procesos compuestos Si **P** y **Q** son procesos entonces **(P || Q)** representa la ejecución concurrente de **P** y **Q**.

```
ITCH = (sratch -> STOP).
CONVERSE = (think -> talk -> STOP).
```

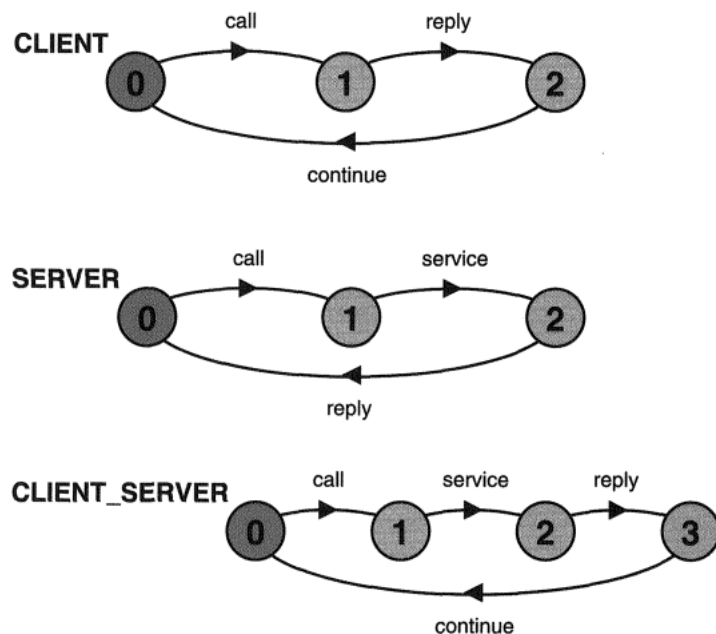
```
||CONVERSE_ITCH = (ITCH || CONVERSE).
```

Relabeling Se aplica a procesos para cambiar el nombre de las etiquetas de acciones. La forma general es **/ { nuevaEtiqueta.1/viejaEtiqueta.1, ..., nuevaEtiqueta.n/viejaEtiqueta.n }**

```
CLIENT = (call -> wait -> continue -> CLIENTE).
```

```
SERVER = (request -> service -> replay -> SERVER).
```

```
|| CLIENT_SERVER = (CLIENT || SERVER) /{call/request, reply/wait}.
```



Etiquetado de procesos Si P es un proceso, entonces $a:P$ prefija, a cada etiqueta del alfabeto de P , la etiqueta a

```
|| TWO_SWITCH = (a:SWITCH || b::SWITCH)
```

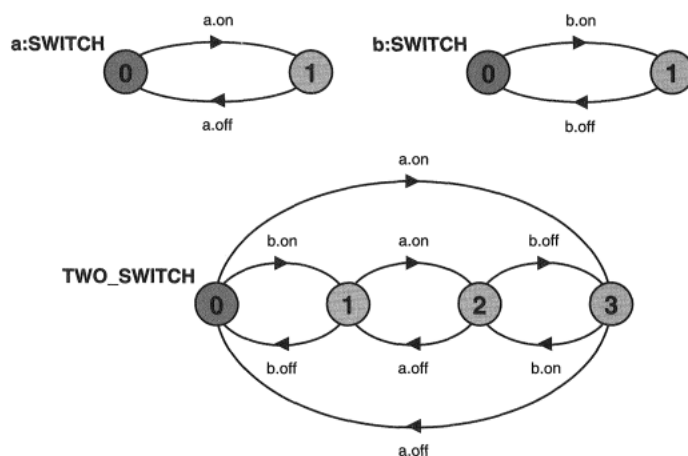


Figure 3.8 Process labeling in TWO_SWITCH.

Generación de etiquetas dentro procesos Si P es un proceso, entonces $a_1, \dots, a_n :: P$ reemplaza cada acción n de P por las etiquetas $a_1.n, \dots, a_x.n$.

$RESOURCE = (acquire \rightarrow release \rightarrow RESOURCE).$

$USER = (acquire \rightarrow use \rightarrow release \rightarrow USER).$

$|| RESOURCE_SHARE = (a:USER || b:USER || \{a,b\} :: RESOURCE)$

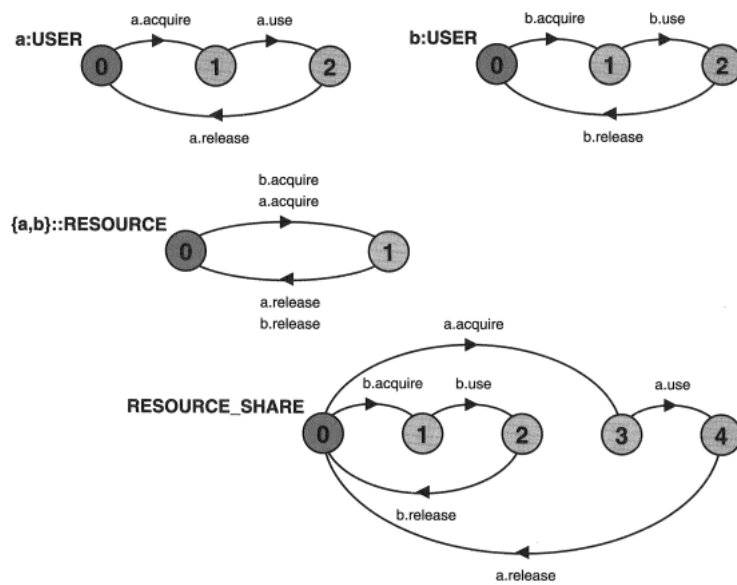


Figura 3.6. Diagrama de estados de RESOURCE_SHARE

Otra forma de escribir esto es:

$|| SWITCHES(N=3) = (\{a,b\}:SWITCH)$

Ocultamiento de etiquetas Cuando se aplica $\backslash\{a_1, a_2\}$ en un proceso P , oculta las etiquetas mencionadas en ese conjunto y convierte las transiciones por estas etiquetas a transiciones por τ .

Cuando se aplica $@\{a_1, a_2\}$ en un proceso P , oculta las etiquetas no mencionadas en ese conjunto y convierte las transiciones por esas etiquetas a transiciones por τ .

Esta definición:

$USER = (acquire \rightarrow use \rightarrow release \rightarrow USER)$

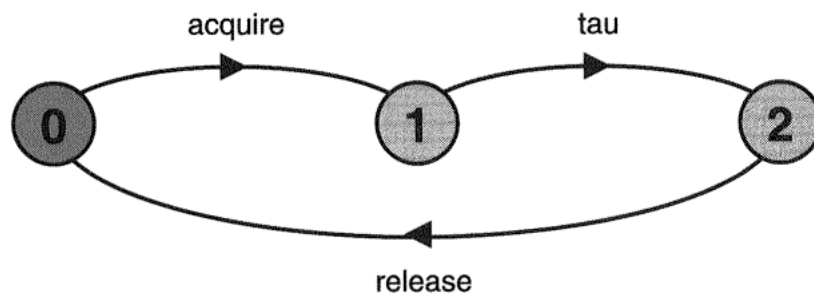
$@\{acquire, release\}.$

y esta:

$USER = (acquire \rightarrow use \rightarrow release \rightarrow USER)$

$\backslash\{use\}.$

son equivalentes y generan:



For all keyword Para cada valor de un rango genera tantas etiquetas / procesos:

Ejemplo:

```
const N = 3
Philosopher = (think -> sit -> left.get -> right.get ->
               eat -> left.put -> right.put -> stand -> Philosopher).
Fork = (get -> put -> Fork).
||DP = (forall [i:0..N] (phil[i]:Philosopher ||
                       {phil[i].left,phil[(i+1)%(N+1)].right}::Fork )).
```

Para correr los tests de progress siempre se asume **fairness**, esto es que hay equiprobabilidad al momento de transicionar no-deterministicamente por las etiquetas.

5. Verificación de propiedades

Chequeo exhaustivo: Se compone el modelo con un proceso de testeo que simula una situación y compara los resultados del modelo con los resultados esperados. Si éste, no responde correctamente el proceso termina en error.

Si, al componer el test con el modelo, hay alguna traza que termina en error entonces ocurrieron alguna de estas dos cosas:

- El test no modela la propiedad deseada y hay que modificarlo.
- El modelo no cumple con la propiedad deseada y hay que modificarlo.

En LTS tenemos la palabra reservada **property**, para indicar que un proceso es una propiedad y debe ser completo. Agrega una transición desde cada estado a un estado de error por cada etiqueta no usada desde el mismo.

5.1. Propiedades de safety

Son las propiedades que nos aseguran que *nunca suceden cosas malas*. Por ejemplo:

- No hay deadlock
- No hay exclusión mutua
- Siempre se preserva el invariante

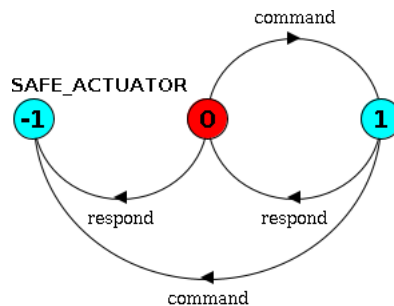
Un sistema **satisface** una propiedad de este tipo si no existe ninguna traza en donde lo malo sucede. Si la **viola**, entonces, podemos mostrar una traza finita en la que esto sucede.

5.1.1. Property

Una **property** es una FSP completo que describe algo que debe pasar y que si no pasa va a ERROR o a STOP.

En todos sus estados todas las acciones del alfabeto están habilitadas. Todas aquellas que no fueron explícitamente declaradas, van a ERROR.

```
property SAFE_ACTUATOR = (command -> respond -> SAFE_ACTUATOR).
```



Si, la composición de una **property** con un sistema, no genera un error o nunca se detiene entonces ese sistema cumple con la propiedad. Llamamos al **automáta observador** del sistema A al FSP generado con **property** que describe una propiedad que debe ser cumplida por A .

Propiedades

- Si S satisface una propiedad safety P entonces $S||T$ también satisface P .
- Si P es un automáta observador y **ERROR** no es alcanzable en $S||P$, entonces **ERROR** tampoco es alcanzable en $(S||P)||Q$, cualquiera sea Q ? (**No queda claro**)

5.2. Propiedades de Liveness

Una propiedad de liveness asegura que *algo bueno va a pasar en algún momento*. Un sistema, no cumple una propiedad de este tipo si existe una traza en donde nunca sucede lo bueno.

Para poder garantizar propiedades de liveness deben realizarse algunas suposiciones:

- **Weak Fairness:** Si a partir de un instante, una acción está habilitada continuamente, entonces no puede ser ignorada.
- **Fair Choice:** Si una elección sobre un conjunto de transiciones es ejecutada infinita veces, entonces cada transición del conjunto será ejecutada infinitas veces.

Propiedad: Si S satisface una propiedad de liveness P entonces $S||T$ también satisface P .

5.2.1. Progress

Una propiedad de progress es una clase particular de liveness. Indica que siempre debe ser posible (eventualmente) ejecutar una o más acciones.

En LTS tenemos la palabra clave **property** para definir este tipo de propiedades. Una vez compilado los LTS, debemos correr un *progress check* para verificar que no sean violadas.

```
COIN =(toss->heads->COIN |toss->tails->COIN).
```

```
progress HEADS = {heads}.
```

```
progress TAILS = {tails}.
```

Conjunto terminal de estados: Conjunto en donde cada estado es alcanzable desde cualquier otro del conjunto y no existen transiciones de un estado del conjunto a un estado fuera del conjunto.

Una propiedad de progreso se viola si es posible encontrar un conjunto terminal en el que ninguna de las acciones de la propiedad está en el conjunto terminal.

```
property USER = (acquire -> use -> release -> USER)
```

5.3. Bisimulación

La bisimulación es una relación de equivalencia (reflexiva, transitiva y simétrica). Separa los autómatas en clases que no dependen de su cantidad de estados y estructura sino por comportamiento.

La idea, es que dos autómatas P y Q son bisimilares si para cada uno de los estados de P , existe un estado equivalente en Q . Osea que tanto en P como en Q , existen estados que soportan las mismas etiquetas y, además, cualquier estado que acción que realicemos lleve a otro par de estados equivalentes.

DEFINICIÓN: Bisimulación fuerte (Strong Bisimulation)

Sea \mathcal{P} el universo de todos los LTS. Una relación binaria $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ es una bisimulación fuerte si y solo si para todo par $(P, Q) \in \mathcal{R}$ vale que para cada acción $a \in Act \cup \{\tau\}$:

- $(P \xrightarrow{a} P') \Rightarrow (\exists Q' \text{ tal que } Q \xrightarrow{a} Q' \wedge (P', Q') \in \mathcal{R})$ y
- $(Q \xrightarrow{a} Q') \Rightarrow (\exists P' \text{ tal que } P \xrightarrow{a} P' \wedge (P', Q') \in \mathcal{R})$

DEFINICIÓN: Bisimilaridad fuerte (Strong bisimilarity)

Dos LTS $P, Q \in \mathcal{P}$ son fuertemente bisimilares ($P \sim Q$) sii \exists una bisimulación fuerte R tal que $(P, Q) \in R$.

$$\sim = \cup \{R | R \text{ es una bisimulación fuerte} \}$$

DEFINICIÓN: Simulación fuerte

Sea \mathcal{P} el universo de todos los LTS. Una relación binaria $R \subseteq \mathcal{P} \times \mathcal{P}$ es una simulación fuerte si y solo si:

$$(P, Q) \in R \Rightarrow (\forall a \in \Sigma \cup \{\tau\} \\ ((P \xrightarrow{a} P') \Rightarrow (\exists Q' \text{ tal que } Q \xrightarrow{a} Q' \wedge (P', Q') \in R)))$$

DEFINICIÓN: Bisimulación débil

Sea \mathcal{P} el universo de todos los LTS. Una relación binaria $R \subseteq \mathcal{P} \times \mathcal{P}$ es una bisimulación débil sii:

$$(P, Q) \in R \Rightarrow \left(\forall a \in \Sigma \cup \{\tau\} \begin{array}{c} ((P \xrightarrow{a} P') \Rightarrow (\exists Q' \text{ tal que } Q \xrightarrow{a} Q' \wedge (P', Q') \in R)) \\ \wedge \\ ((Q \xrightarrow{a} Q') \Rightarrow (\exists P' \text{ tal que } P \xrightarrow{a} P' \wedge (P', Q') \in R)) \end{array} \right)$$

Donde:

$$\xrightarrow{a} = \begin{cases} (\xrightarrow{\tau})^* \circ \xrightarrow{a} \circ (\xrightarrow{\tau})^* & \text{si } a \neq \tau \\ (\xrightarrow{\tau})^* & \text{si } a = \tau \end{cases}$$

DEFINICIÓN: Bimimilaridad débil

Dos LTS $P, Q \in \mathcal{P}$ son débilmente bisimilares ($P \approx Q$) sii \exists una bisimulación débil R tal que $(P, Q) \in R$.

$$\approx = \cup \{R \mid R \text{ es una bisimulación débil}\}$$

5.3.1. El juego de la bisimilaridad

Juego de la Bisimilaridad fuerte Sean P y Q dos LTS, el juego consiste en un **atacante** y un **defensor**. El atacante intenta demostrar que P y Q no son bisimilares y, el defensor, que lo son.

En cada ronda, los jugadores cambian la configuración del juego cambiando el estado de uno de los autómatas de la siguiente manera:

- El atacante elige uno de los procesos en la configuración corriente y hace una movida por alguna transición con etiqueta a en alguno de los procesos.
- El defensor debe responder haciendo una movida por una transición etiquetada con a en el proceso.

Si el juego es infinito, entonces el defensor gana y P y Q son bisimilares. Si, en algún momento, el defensor no puede realizar la acción que realiza el atacante entonces el juego termina, gana el atacante y P y Q no son bisimilares.

Si son bisimilares, entonces hay que dar la relación de bisimilaridad. Para esto, hay que conseguir todos los pares de estados que pertenecen a ella, osea, hay que conseguir todos los pares $(p, q) \in P \times Q$ que pueden ser alcanzados por una etiqueta.

Juego de la Bisimilaridad débil: Idem anterior, pero ahora el defensor puede moverse por \Rightarrow y, el atacante, por a lo sumo, un τ (o no, la verdad esto no queda claro. Los tipos nunca no nos supieron decir así que, si alguien se entera, por favor, avise).

Parte II

Lógica Temporal Lineal

6. Estructuras de Kripke

DEFINICIÓN: Estructura de Kripke

Una **estructura de Kripke** es un par (W, R) tal que:

- W es un conjunto de nodos,
- R es una relación binaria sobre W ($R \subseteq W \times W$).

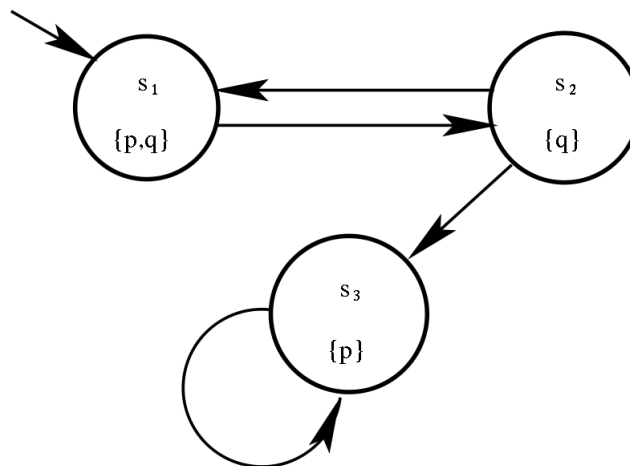
O sea que es **es un grafo**.

DEFINICIÓN: Función de evaluación

Una **función de evaluación** $v : Pr \times W \rightarrow \{T, F\}$ asigna a cada proposición un valor de verdad en cada estado del posible.

Las estructuras de Kripke, junto con una función de evaluación nos permite representar un programa en base a las propiedades que se deben cumplir en cada uno de sus estados. Cada estado es un nodo y se une con flechas a todos los estados a los que puede pasar el programa desde ese estado. La función de evaluación, nos indica que propocisiones son validad en cada momento de la ejecución.

Ejemplo de Modelo con Krikpe



El kripke representa un programa de tres estados $W = \{s_1, s_2, s_3\}$ en el que pueden valer dos proposiciones p y q . En cada estados, marcamos cuales de ellas vale entre llaves. Osea que en s_1 valen las dos, en s_2 vales solo q y en s_3 solo p .

DEFINICIÓN: Trazas

Una traza de una estructura de Kripke son una secuencia infinita de estados $\sigma = \{\sigma_0, \sigma_1, \dots\}$ que representa la ejecución de un programa. σ_i representa el estado al que llegó el programa en el i -ésimo paso.

Sean ψ una fórmula de LTL:

7. Lógica temporal lineal (LTL)

- Variables proposicionales: p, q, r, \dots
- Conectivos lógicos clásicos:
 - Conjunción ($\psi_1 \wedge \psi_2$)
 - Negación ($\neg\psi$)
 - Disyunción ($\psi_1 \vee \psi_2$)
 - Implicación ($\psi_1 \implies \psi_2$)
- Operadores modales:
 - Siempre ($\Box\psi$ ó $G\psi$)
 - En el próximo paso ($X\psi$)
 - Eventualmente ($\Diamond\psi$ ó $F\psi$)
 - Hasta que ($\psi_1 \text{ U } \psi_2$)

Nos va a servir para hablar sobre propiedades de alguna ejecución de un programa, es decir, sobre propiedades de alguna de las trazas de un modelo de Kripke.

7.1. Semántica

Dada una estructura de Kripke (W, R) , una función de evaluación v y σ una traza del Kripke decimos que un estado w satisface la fórmula ψ si ψ es verdadera en ese estado. Sea p una proposición de la lógica modal, entonces:

- $\sigma[i] \models p$ si y solo si $v(p, \sigma[i])$
- $\sigma[i] \models \neg\psi$ si y solo si $\sigma[i] \not\models \psi$
- $\sigma[i] \models \psi_1 \wedge \psi_2$ si y solo si $\sigma[i] \models \psi_1$ y $\sigma[i] \models \psi_2$.
- $\sigma[i] \models \psi_1 \vee \psi_2$ si y solo si $\sigma[i] \models \psi_1$ o $\sigma[i] \models \psi_2$.
- $\sigma[i] \models \psi_1 \implies \psi_2$ si y solo si $\sigma[i] \not\models \psi_1$ o $\sigma[i] \models \psi_2$
- $\sigma[i] \models \neg\psi$ si y solo si $\sigma[i] \not\models \psi$
- $\sigma[i] \models X\psi$ si y solo si $\sigma[i+1] \models \psi$
- $\sigma[i] \models \Diamond\psi$ si y solo si $(\exists j : i \leq j) \sigma[j] \models \psi$
- $\sigma[i] \models \Box\psi$ si y solo si $(\forall j : i \leq j) \sigma[j] \models \psi$
- $\sigma[i] \models \psi_1 \text{ U } \psi_2$ si y solo si $(\exists k : i \leq k)(\sigma[k] \models \psi_2 \wedge ((\forall j : i \leq j < k) \sigma[j] \models \psi_1))$

Decimos que una traza σ satisface una fórmula ψ ($\sigma \models \psi$) si y solo si $\sigma[0] \models \psi$.

8. Verificación de propiedades

Decimos que un modelo M (Kripke) satisface un fórmula ψ de LTL si y solo si para toda traza σ de M , vale $\sigma \models \psi$.

8.1. Autómata de buchi

DEFINICIÓN: Autómata de Büchi

Son autómatas de estados finitos que reconocen lenguajes de palabras infinitas. Solo aceptan una cadena si pasa infinitas veces por uno o más estados de aceptación. Los vamos a usar para representar conjuntos de trazas de un sistema que cumplan ciertas propiedades.

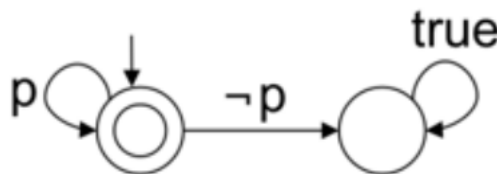


Figura 1: Autómata de buchi para la fórmula $\Box p$. Se mantiene en un estado de aceptación mientras valga p . En el primer momento que no lo hace, va al segundo estado y se queda ciclando en el mismo (sin importar el valor actual de p).

8.2. Método de verificación

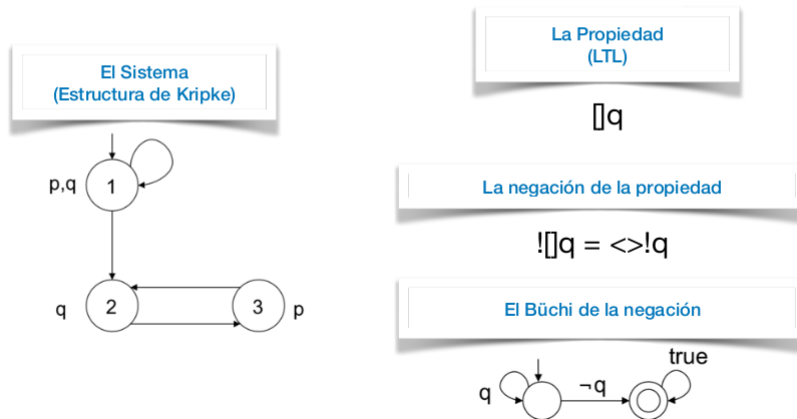
Para comprobar que una fórmula ψ de la lógica temporal es satisfecha por una estructura de Kripke M debemos hacer lo siguiente:

1. Convertir la formula $\neg\psi$ en un autómata de Büchi $A\psi$ (No dijeron como hacer esto, hacerlo por intuición y rogar que este bien).
2. Convertir el Kripke M en un autómata de Büchi AM . Todos los estados del autómata generado son de aceptación.
3. Componer AM con AP (los autómatas de los dos pasos anteriores)
4. Verificar si el lenguaje que acepta la composición es vacío. Debemos buscar un ciclo que contenga un estado de aceptación y sea alcanzable desde el estado inicial.
 - Si el lenguaje es vacío entonces se cumple ψ .
 - Si no lo es, existe una traza en M que cumple la negación de ψ y funciona como contra ejemplo.

Ejemplo:

Paso 1

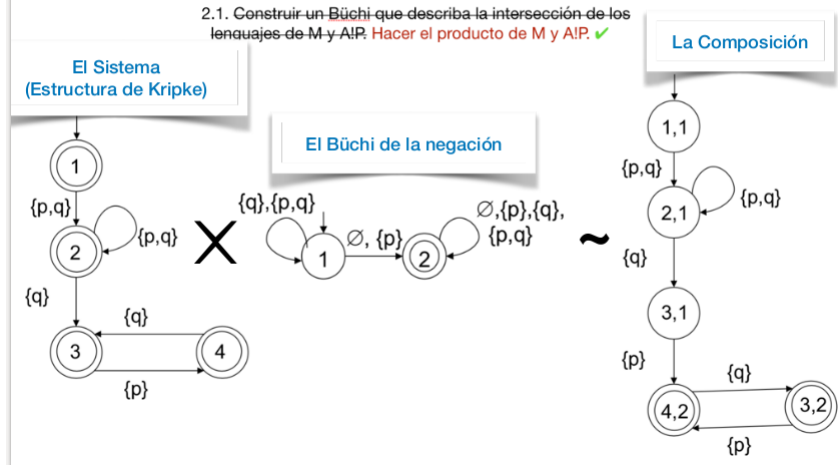
1. Convertir la formula LTL IP a un automata **büchi** AIP que caracteriza todas las trazas que satisfacen IP ✓



Paso 2.1

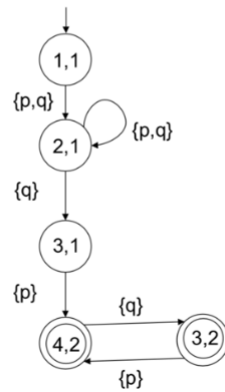
2. Chequear que si las trazas de M son disjuntas con las trazas de $A|P$

2.1. Construir un Büchi que describa la intersección de los lenguajes de M y A!P. Hacer el producto de M y A!P. ✓



Paso 2.2

2.2. Verificar si tiene un lenguaje vacío ✓



Componente fuertemente conexas con estado de aceptación: $\{(4,2), (3,2)\}$



El lenguaje es no vacío.
La palabra $\{p,q\} \{q\} \{p\} \{q\} \{p\}^\omega$ es
aceptada por el camino $(1,1)(2,1)(3,1)((4,2)(3,2))$



Existe una palabra aceptada por el sistema y
también por la negación de la propiedad



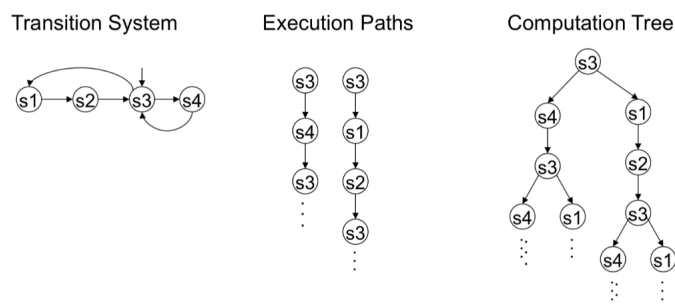
El sistema viola la propiedad $\Box q$

Parte III

Lógica Computacional Árborea (CTL)

9. Sintaxis y semántica

A diferencia de LTL, que se concentra en una única traza, nos va a permitir describir propiedades sobre la totalidad de las trazas de un sistema. Dejamos de verlas como caminos desconexos para pasar a representarlas como arboles de ejecución.



Agrega las cuantificaciones A y E a los operadores modales, lo que permite, asertar cosas sobre todos los caminos que derivan de un nodo:

- A: Para todo camino
- E: Existe un camino

DEFINICIÓN: Función de evaluación L

Dado un conjunto de proposiciones atómicas AP y una estructura de Kripke $M = (W, R)$ sobre AP , la función $L : W \rightarrow 2^{AP}$ toma un estado de M y devuelve el conjunto de proposiciones que valen en ese estado (la imagen de la función es el conjunto de parte de AP).

9.1. Semántica

Dado un árbol de cómputo infinito satisface una formula si su raíz s_0 lo hace. Sean ψ y ψ_1 dos fórmulas de CTL:

- $s \models p$ si y solo si $v(p, s)$ (con p una propoción atómica)
- $s_0 \models EX \psi$ si y solo si existe un camino $S = \{s_0, \dots\}$ tal que $S[1] \models \psi$
- $s_0 \models AX \psi$ si y solo si para todo camino S de la forma $\{s_0, \dots\}$ vale que $S[1] \models \psi$
- $s_0 \models EG \psi$ si y solo si existe un camino $S = \{s_0, \dots\}$ en el que siempre vale ψ .

- $s_0 \models \text{AG } \psi$ si y solo si para todo camino S de la forma $\{s_0, \dots\}$ siempre vale ψ ($\forall i \geq 0$ vale $S[i] \models \psi$).
- $s_0 \models \text{EF } \psi$ si y solo si existe un camino $S = \{s_0, \dots\}$ en el que, en algún momento, vale ψ ($\exists i \geq 0$ tal que $S[i] \models \psi$).
- $s[0] \models \text{AF } \psi$ si y solo si para todo camino S de la forma $\{s_0, \dots\}$ vale, en algún momento, ψ ($\exists i \geq 0$ tal que $S[i] \models \psi$).
- $s_0 \models \psi \text{ EU } \psi_1$ si y solo si existe un camino $S = \{s_0, \dots\}$ tal que
 $(\exists i \geq 0) (S[i] \models \psi_1 \wedge (\forall 0 \leq j \leq i) S[j] \models \psi))$.
- $s_0 \models \psi \text{ AU } \psi_1$ si y solo si para todo camino S de la forma $\{s_0, \dots\}$ vale que
 $(\exists i \geq 0) (S[i] \models \psi_1 \wedge (\forall 0 \leq j \leq i) S[j] \models \psi))$.

9.1.1. Equivalencias

- $\text{AX } \psi \equiv \neg \text{EX } (\neg \psi)$
- $\text{AG } \psi \equiv \neg \text{EF } (\neg \psi)$
- $\text{EF } \psi \equiv (\text{true EU } \psi)$
- $\text{AF } \psi \equiv \neg \text{EG } (\neg \psi)$
- $\psi \text{ AU } \psi_1 \equiv \text{AF } \psi_1 \wedge \neg(\neg \psi_2 \text{ EU } \neg(\psi \vee \psi_2))$

10. Verificación explícita de propiedades

Dado una estructura de Kripke, deseamos determinar si su árbol de cómputo cumple con una fórmula ψ de CTL. Para esto, debemos representar las fórmulas como conjuntos de estados.

10.1. Fórmulas como conjuntos característicos

DEFINICIÓN: Conjunto característico

Dados un modelo $M = \langle (W, R), v \rangle$ sobre un conjunto de proposiciones atómicas (AP) y una fórmula de CTL ϕ . El conjunto característico $\llbracket \phi \rrbracket_M$ de ϕ sobre M es el conjunto de estados en los que ϕ es verdadera.

Semántica de CTL vista como conjuntos característicos

- $\llbracket a \rrbracket_M = \{s \in W \mid v(s, a) = \text{True}\}$ para $a \in AP$
- $\llbracket \neg\psi \rrbracket_M = W - \llbracket \psi \rrbracket_M$
- $\llbracket \psi_1 \vee \psi_2 \rrbracket_M = \llbracket \psi_1 \rrbracket_M \cup \llbracket \psi_2 \rrbracket_M$
- $\llbracket \psi_1 \wedge \psi_2 \rrbracket_M = \llbracket \psi_1 \rrbracket_M \cap \llbracket \psi_2 \rrbracket_M$
- $\llbracket \text{EX } \psi \rrbracket_M = \{s \in W \mid \exists (s, t) \in R \text{ tal que } t \in \llbracket \psi \rrbracket_M\}$
- $\llbracket \text{EG } \psi \rrbracket_M = \{s \in W \mid \text{existe un camino } S = \{s, \dots\} \text{ tal que para todo estado } S[i] \text{ con } i \geq 0, S[i] \in \llbracket \psi \rrbracket_M\}$
- $\llbracket \psi \text{ EU } \psi_2 \rrbracket_M = \{s \in W \mid \text{existe un camino } S = \{s, \dots\} \text{ tal que para algún estado } S[i] \text{ con } i \geq 0, S[i] \in \llbracket \psi_2 \rrbracket_M \text{ y para todo } k < i, S[k] \in \llbracket \psi \rrbracket_M\}$

Decimos que un modelo M satisface ψ ($M \models \psi$) si y solo si su estado inicial s_0 la satisface ($s_0 \in \llbracket \psi \rrbracket_M$).

10.1.1. Algoritmo de verificación explícito:

Para **EX** ψ :

1. Conseguir todos los estados en los que vale ψ .
2. Hacer un paso hacia atrás en cada uno de ellos. (Todos estos conjuntos son los que cumplen la fórmula)

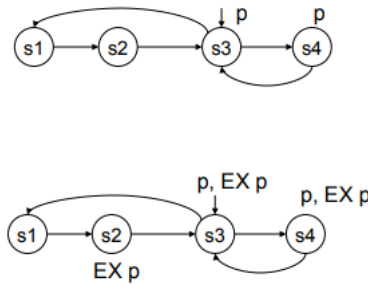


Figura 2: Cómputo de EX

Para ψ **EU** ψ_1 :

1. Conseguir todos los estados en los que vale ψ_1 (estos estados cumplen la fórmula)
2. Ir hacia atrás mientras valga ψ (cada estado por el que pasamos cumple la fórmula).

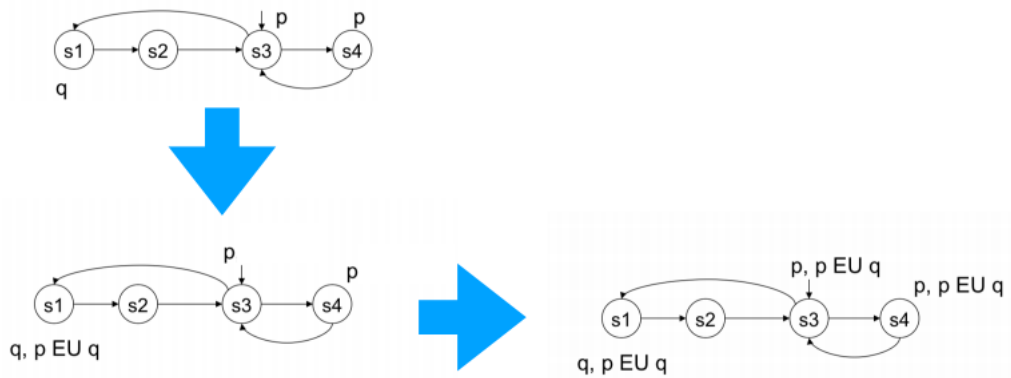


Figura 3: Cómputo de EU

Para EG ψ :

1. Eliminar todos los estados en donde no vale ψ y conectar los nodos que quedan. Osea, que hay que remplazarlo por caminos entre cada uno de sus nodos de entrada a todos sus nodos de salida.
2. Computar las componentes fuertemente conexas.
3. Marcar todos los estados del Kripke en los que valen p que llegan a alguna componente fuertemente conexas.

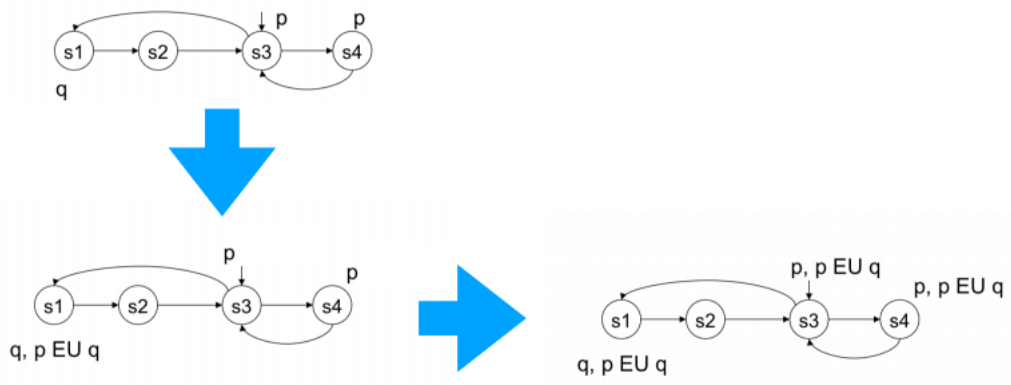


Figura 4: Cómputo de EG

Para el resto de las fórmulas: Convertirlas a fórmulas que solo usen EX, EG y EU usando las equivalencias de la sección 9.1.1. Y aplicar alguno de los algoritmos anteriores.

11. Verificación implícita de propiedades

El algoritmo visto en la sección anterior, utiliza distintas técnicas para recorrer grafos, por lo que se necesita una representación explícita del espacio de estados del modelo. Esto limita los sistemas que podemos representar ya que, por ejemplo, en sistemas concurrentes la cantidad de estados aumenta exponencialmente con la cantidad de sub-procesos compuestos.

Por esta razón, se desarrolló lo que se llama la verificación simbólica de propiedades. Esta técnica utiliza diagramas de binarios de decisión para representar el sistema como una fórmula proposicional lógica que podemos conjugar con la propiedad que deseamos verificar.

La fórmula obtenida describe los estados del modelo en los que la propiedad vale.

11.1. Árboles binarios de decision

Si bien, las fórmulas booleanas/LTL que tratamos en clase no tienen muchas proposiciones y están bastante sintetizadas, en la vida real no es así. En muchos casos, tendremos demasiadas variables y no será fácil escribir una formula sintácticamente compactas, es decir, nos puede pasar que haya operaciones innecesarias.

Resolver esta fórmula, implicaría evaluar toda las proposiciones que están en esta parte innecesaria, por lo menos, una vez y ocupar tiempo de computo en operaciones booleanas sin sentido. Si bien, evaluarla una vez, no parece que tenga mucho impacto, hacer esto para cada estado de un Kripke implicaría demasiado tiempo.

Pongamos un ejemplo: Si a y b son las variables proposicionales de nuestro modelo, sea $\psi = (\neg a \wedge b) \vee (a \wedge b)$. Vemos que $(\neg a \wedge b) \vee (a \wedge b)$ es equivalente a b . Osea que en cada estado que evaluemos ψ , estaríamos evaluando innecesariamente dos veces a y a b una vez de más.

Árboles de decisión binarios (BDT): Son estructuras que pueden ser usados para representar y manipular fórmulas booleanas. La cantidad de nodos de un BDT es $2^n - 1$, con n la cantidad de variables proposicionales de la fórmula representada.

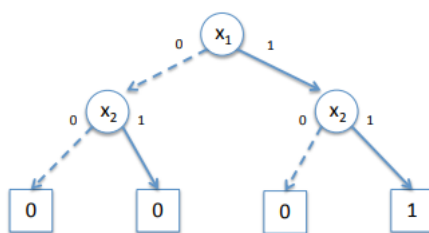


Figura 5: Árbol de decisión binaria para $x_1 \wedge x_2$

Diagramas de decisión binaria (BDD): Son una representación más compacta de los BDT. Eliminan la representación redundante de variables booleanas y permite compartir sub-árboles

iguales.

Son representaciones canónicas, es decir, dada dos fórmulas semanticamente equivalentes, si damos un orden a sus variables y creamos sus BDD para cada una de ellas entonces, los BDD, serán iguales. Osea que, en el ejemplo anterior, ψ y b tienen el mismo árbol. De esta forma, evitamos realizar evaluaciones de más.

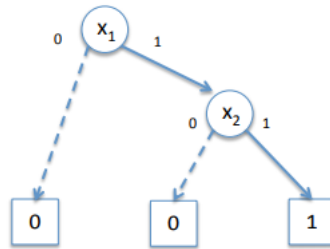


Figura 6: Diagrama de decisión binaria para $x_1 \wedge x_2$. El nodo x_2 del lado izquierdo era redundante porque la fórmula da siempre 0 sin importar su valor.

11.1.1. Construcción de un ROBDD

Dada una fórmula queremos conseguir un OBDD reducido para la misma:

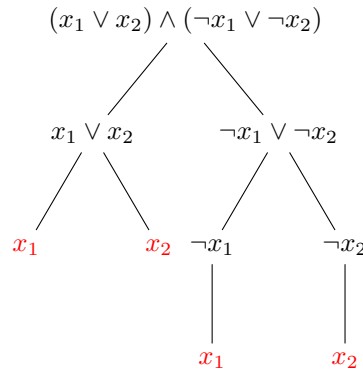
1. Dividir la fórmula en sub-fórmulas hasta llegar a las variables proposicionales.
2. Crear un ROBDD para cada proposición e ir combinandolos para obtener cada sub-fórmula.
Luego, para cada ROBDD, crearlos e ir componiendolos hasta llegar a la fórmula

Algoritmo de composición de ROBDD : Dados dos ROBDD A y B que representan las fórmulas ϕ y ψ , respectivamente que hayan sido creados con el mismo orden de variables:

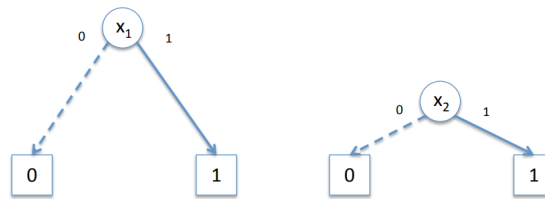
1. Elegir un ROBDD por el que empezar y tomar la raíz.
2. Si la variable representada por la raíz no está en el otro OBDD entonces agregarla como raíz, con ambos caminos apuntando hacia la raíz original.
3. Elegir un camino y seguirlo hasta el próximo nivel en ambos ROBDD. Si los niveles no representan las mismas variables, entonces agregar un nodo hipotético entre ambos niveles. Ambos caminos del nodo hipotético van al nivel más bajo.
4. Si llegamos a las hojas, conseguimos un valor para ϕ y uno para ψ , aplicarles la operación lógica que queremos conseguir.
5. Agregar el camino recorrido (con los nodos hipotéticos) a un nuevo árbol y poniendo como hoja el resultado de la operación. Y eliminando ocurrencias redundantes (nodos cuyos caminos van al mismo nodo).

6. Volver a repetir para cada camino posible.

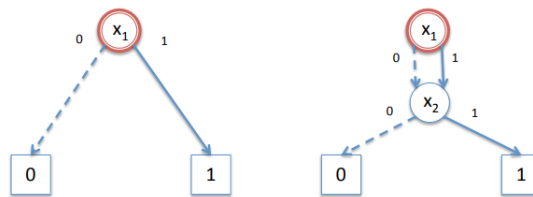
Ejemplo: Queremos conseguir el ROBDD de $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$, el árbol sintáctico de la fórmula es:



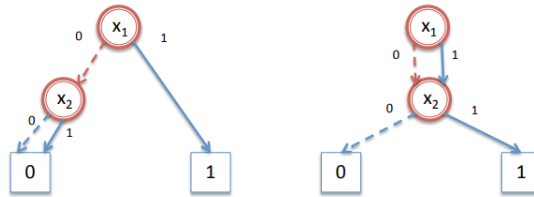
Osea que, para empezar, debemos construir los OBDDs para x_1 y x_2 :



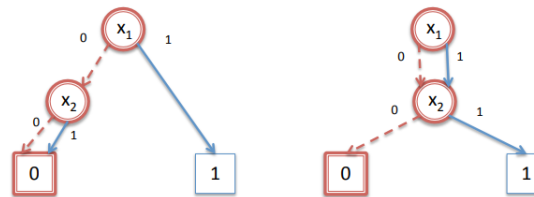
Para conseguir el OBDD de $x_1 \vee x_2$, debemos recorrer los dos árboles asignando, en cada nivel, un valor a la variable de ese nivel. Empezamos por el de x_1 . Como en el árbol de la derecha no existe, le agregamos un nodo hipotético como raíz.



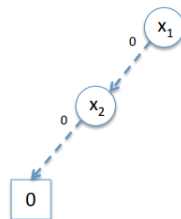
Elegimos el camino $x_1 = 0$ y avanzamos al siguiente nivel que, en el lado derecho, corresponde a x_2 y falta en el izquierdo:



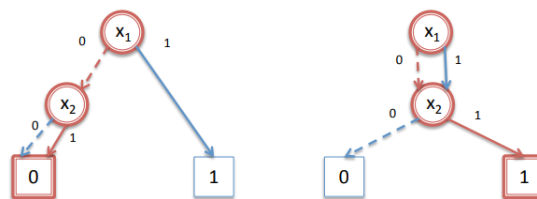
Y elegimos el camino $x_2 = 0$:



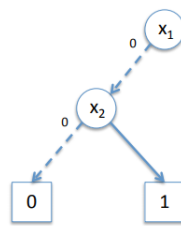
Conseguimos el valor 0 en ambos árboles. $0 \vee 0$ es 0, entonces creamos un nuevo árbol con el camino que acabamos de recorrer y el valor 0 como hoja:



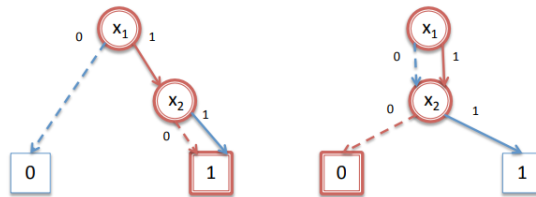
Ahora, subimos un nivel, y elegimos $x_2 = 1$:



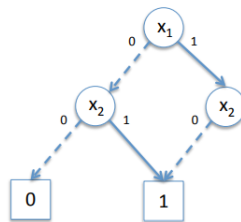
Conseguimos los valores 0 y 1. Combinandolos con \vee obtenemos 1, entonces agregamos ese camino al árbol que estamos creando:



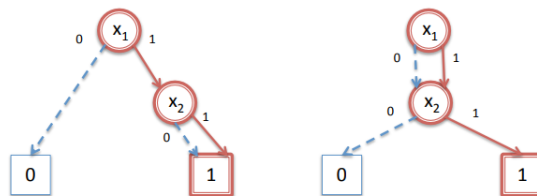
Como ya consideramos todos los posibles valores de x_2 para cuando $x_1 = 0$, subimos dos niveles y vemos el caso $x_1 = 1$ y $x_2 = 0$:



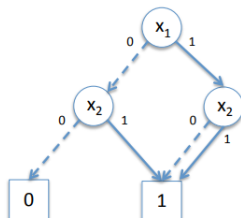
Es el mismo caso que el anterior, agregar el camino y terminarlo en 1:



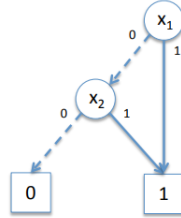
Analizamos el único camino que falta ($x_1 = 1$ y $x_2 = 2$):



Ambos valores son 1 y $1 \vee 1 = 1$: i



Vemos que cuando $x_1 = 1$, el nodo de x_2 es redundante, pues ambos caminos van a 1. Entonces, lo eliminamos:



Para crear los ROBDD para la negación, solo hay que invertir los valores de las hojas del ROBDD de la fórmula que queremos negar. Luego combinamos ambos ROBDDs de la misma forma para obtener $\neg x_1 \vee \neg x_2$.

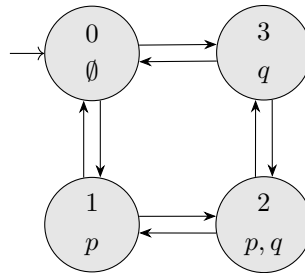
Por último, usando el mismo algoritmo, combinamos los ROBDDs de $\neg x_1 \vee \neg x_2$ y $x_1 \vee x_2$ para conseguir la fórmula original.

11.2. Cálculo de EX ψ

Ya sabemos como caracterizar los estados de un Kripke que cumplen con una fórmula CTL ψ y, además, tenemos una buena forma de representar y evaluar fórmulas booleanas (los OBDDs). Lo siguiente será definir un algoritmo que nos permita encontrar una fórmula booleana que describa los estados que satisfacen ψ .

Dado un modelo $\mathcal{M} = \langle (W, R), v \rangle$ y una fórmula ψ , queremos obtener una fórmula que podamos evaluar en cada estado del Kripke que devuelva *true* cuando ese estado cumple EX ψ .

Vamos a ir haciendolo con un ejemplo, supongamos que tenemos el siguiente Kripke cuyas proposiciones son p y q :



11.2.1. Conversión de un Kripke a fórmulas booleanas

Una forma de describir un Kripke, es usando un estado inicial y un conjunto de transiciones. Si logramos representar cada uno de ellos como una fórmula booleana, ganamos.

1. Describir el estado inicial como una fórmula booleana I . Esto es fácil, a cada estado inicial lo podemos representar como la conjunción de todas las variables proposicionales que valen en ese estado con la negación de todas las que no valen.

Si hay más de un estado inicial, se hace la disyunción de todas sus fórmulas.

En nuestro caso, solo tenemos un estado inicial en el que no vale ninguna de las proposiciones del modelo, entonces $I = \neg q \wedge \neg p$.

2. Describir las transición del Kripke como una fórmula booleana R . Para esto, necesitamos hacer una conyunción de las fórmulas de cada transición.

La fórmula de una transición debe describir el estado actual de las variables y el estado que tendrá en el próximo paso. Para indicar el segundo caso, primamos las variables, es decir, les agregamos un apostrofe.

Nombre R_{ij} a la transición que va de el estado i al j , entonces tenemos:

$$\begin{array}{ll} R_{01} = \neg p \wedge \neg q \wedge \neg q' \wedge p' & R_{21} = p \wedge q \wedge \neg q' \wedge p' \\ R_{03} = \neg p \wedge \neg q \wedge q' \wedge \neg p' & R_{23} = p \wedge q \wedge q' \wedge \neg p' \\ R_{10} = p \wedge \neg q \wedge \neg q' \wedge \neg p' & R_{30} = \neg p \wedge q \wedge \neg q' \wedge \neg p' \\ R_{12} = p \wedge \neg q \wedge q' \wedge p' & R_{32} = \neg p \wedge q \wedge q' \wedge p' \end{array}$$

$$R = R_{01} \vee R_{03} \vee R_{10} \vee R_{12} \vee R_{21} \vee R_{23} \vee R_{30} \vee R_{32}$$

Con I y R nos alcanza para describir el Kripke $K = I \wedge R$.

11.2.2. Eliminación del existencial

En la sección 10.1, vimos que EX ψ es el conjunto de estados $\{s \in W \mid \exists (s, t) \in R \wedge t \in \llbracket \psi \rrbracket_M\}$. Tenemos que pensarlo como fórmula booleana que describa estados del Kripke.

$$\text{EX}\psi \equiv \exists V' \cdot R \wedge \psi[V'/V]$$

Esto es, si aplicamos EX ψ a un estado, entonces ese estado lo cumple si existe un conjunto V' de variables proposicionales tal que vale R (hay una transición desde ese estado al nuevo) y $\psi[V'/V]$ (si reemplazamos las variables proposicionales de V por sus primadas, entonces vale ψ). El $\exists V'$ se traduce en un existencial por cada variable del conjunto de proposiciones.

Seguimos sin tener una fórmula proposicional, debemos eliminar el existencial, para esto hay que aplicar la siguiente equivalencia. Si f y v una proposición:

$$\exists v \cdot f = f[\text{True}/v] \vee f[\text{False}/v]$$

Seguimos con el ejemplo: Supongamos que $\psi = p \wedge q$, queremos computar $\text{EX } \psi$.

$$\begin{aligned}
\text{EX } \psi &= \exists V' \cdot R \wedge \psi[V'/V] \\
&= \exists V' \cdot R \wedge (p' \wedge q') \\
&= \exists p' \cdot (\exists q' \cdot R \wedge (p' \wedge q')) \\
&= \exists p' \cdot (R \wedge (p' \wedge q'))[\text{True}/q'] \vee (R \wedge (p' \wedge q'))[\text{False}/q'] \\
&= \exists p' \cdot (R[\text{True}/q'] \wedge (p' \wedge q')[\text{True}/q']) \vee (R[\text{False}/q'] \wedge (p' \wedge q')[\text{False}/q']) \\
&= \exists p' \cdot (R[\text{True}/q'] \wedge \underbrace{(p' \wedge \text{True})}_{p'}) \vee (R[\text{False}/q'] \wedge \underbrace{(p' \wedge \text{False})}_{\text{False}}) \\
&= \exists p' \cdot (R[\text{True}/q'] \wedge p') \\
&= (R[\text{True}/q'] \wedge p')[\text{True}/p'] \vee (R[\text{True}/q'] \wedge p')[\text{False}/p'] \\
&= (R[\text{True}/q'][\text{True}/p'] \wedge p'[\text{True}/p']) \vee (R[\text{True}/q'][\text{False}/p'] \wedge p'[\text{False}/p']) \\
&= (R[\text{True}/q'][\text{True}/p'] \wedge \text{True}) \vee (R[\text{True}/q'][\text{False}/p'] \wedge \text{False}) \\
&= R[\text{True}/q'][\text{True}/p'] \\
&= (R_{01} \vee R_{03} \vee R_{10} \vee R_{12} \vee R_{21} \vee R_{23} \vee R_{30} \vee R_{32})[\text{True}/q'][\text{True}/p'] \\
&= \underbrace{(R_{01}[\text{True}/q'] \vee R_{03}[\text{True}/q'] \vee R_{10}[\text{True}/q'] \vee R_{12}[\text{True}/q'])}_{\text{False}} \\
&\quad \vee \underbrace{(R_{21}[\text{True}/q'] \vee R_{23}[\text{True}/q'] \vee R_{30}[\text{True}/q'] \vee R_{32}[\text{True}/q'])}_{\text{False}}[\text{True}/p'] \\
&= ((R_{03}[\text{True}/q'] \vee R_{12}[\text{True}/q'] \vee R_{23}[\text{True}/q'] \vee R_{32}[\text{True}/q'])[\text{True}/p']) \\
&= \underbrace{R_{03}[\text{True}/q'][\text{True}/p'] \vee R_{12}[\text{True}/q'][\text{True}/p']}_{\text{False}} \vee \underbrace{R_{23}[\text{True}/q'][\text{True}/p'] \vee R_{32}[\text{True}/q'][\text{True}/p']}_{\text{False}} \\
&= R_{12}[\text{True}/q'][\text{True}/p'] \vee R_{32}[\text{True}/q'][\text{True}/p'] \\
&= (p \wedge \neg q \wedge q' \wedge p')[\text{True}/q'][\text{True}/p'] \vee (\neg p \wedge q \wedge q' \wedge p')[\text{True}/q'][\text{True}/p'] \\
&= (p \wedge \neg q \wedge \text{True} \wedge \text{True}) \vee (\neg p \wedge q \wedge \text{True} \wedge \text{True}) \\
&= (p \wedge \neg q) \vee (\neg p \wedge q)
\end{aligned}$$

Entonces $(p \wedge \neg q) \vee (\neg p \wedge q)$ describe los estados de este autómata que satisfacen la fórmula $\text{EX } \psi$.

11.3. Algoritmo de punto fijo

Al resto de las fórmulas, las describimos de manera recursiva con EX y AX . Esto nos va a permitir utilizar los algoritmos de punto fijo para calcular los conjuntos de estados que satisfagan

ciertas fórmulas.

11.3.1. Los puntos fijos de un reticulado

DEFINICIÓN: Punto fijo

Dada una función $f : D \rightarrow D$, $x \in D$ es un punto fijo de f , si y solo si $f(x) = x$

DEFINICIÓN: Función monótona

Una función $f : D \rightarrow D$ es monótona si y solo si para todo $x, y \in D$, $x \leq y \implies f(x) \leq f(y)$

DEFINICIÓN: Reticulado

Un reticulado es un conjunto S con una relación binaria $\mathcal{R} \subseteq S \times S$ (reflexiva, transitiva y antisimétrica) que define un orden parcial sobre sus elementos.

La relación, además, define un único **supremo** y un único **ínfimo** para cada par $(a, b) \in R$, es decir, vale que $a > b$ ó $b < a$.

En ingeniería notamos $a \vee b$ al supremo entre ambos valores y $a \wedge b$ al ínfimo.

DEFINICIÓN: Reticulado completo

Un reticulado es completo si cumple que para todo subconjunto de S existe un supremo y un ínfimo.

En nuestro caso, los operadores de ínfimo y supremos van a estar dados por la intersección y la unión, respectivamente.

Por ejemplo, dado un conjunto de partes P , podemos definir su supremo como $\bigcup\{s | s \in P\} = P$ y su ínfimo como $\bigcap\{s | s \in P\} = \emptyset$.

Para mostrar los siguientes teoremas vamos a asumir que D es un conjunto de partes.

Teorema 1: Dado un reticulado completo (D, \subseteq) y una función $f : D \rightarrow D$ monótona, entonces su mínimo punto fijo es $\bigcap\{y | f(y) \subseteq y\}$

Notar que estamos definiendo f como una función monótona sobre conjuntos. En la definición de más arriba, habría que cambiar \leq por \subseteq

DEFINICIÓN: Función \cup -continua

Una función monótona $f : D \rightarrow D$ con D finito es \cup -continua si dado un conjunto de conjuntos que cumple $p_1 \subseteq p_2 \subseteq \dots$ implica que $f(\bigcup_i p_i) = \bigcup f(p_i)$

Teorema 2: Dado un reticulado completo (D, \subseteq) y una función monótona \cup -continua $f : D \rightarrow D$ entonces el límite de la secuencia $\bigcup_i f^i(\emptyset) = f(\emptyset) \cup f(f(\emptyset)) \cup f(f(f(\emptyset))) \cup \dots$ es el mínimo punto fijo.

Teorema 3: Dado un reticulado completo (D, \subseteq) y una función monótona $f : D \rightarrow D$ entonces existe un j tal que $F^j(\emptyset) = F^{j+1}(\emptyset)$ y $F^j(\emptyset)$ es el mínimo punto fijo.

Teorema 4: Dado un reticulado completo (D, \subseteq) y una función monótona $f : D \rightarrow D$ entonces el límite de la secuencia $\bigcap_i f^i(S) = f(S) \cup f(f(S)) \cup f(f(f(S))) \cup \dots$ es el máximo punto fijo.

11.3.2. Operadores CTL definidos recursivamente

- | | |
|---|---|
| ▪ $AG\ p = p \wedge AX\ AG\ p$ | ▪ $\llbracket AG\ p \rrbracket = \llbracket p \rrbracket \cap \llbracket AX\ AG\ p \rrbracket$ |
| ▪ $EG\ p = p \wedge EX\ EG\ p$ | ▪ $\llbracket EG\ p \rrbracket = \llbracket p \rrbracket \cap \llbracket EX\ EG\ p \rrbracket$ |
| ▪ $AF\ p = p \vee AX\ AF\ p$ | ▪ $\llbracket AF\ p \rrbracket = \llbracket p \rrbracket \cup \llbracket AX\ AF\ p \rrbracket$ |
| ▪ $EF\ p = p \vee EX\ EF\ p$ | ▪ $\llbracket EF\ p \rrbracket = \llbracket p \rrbracket \cup \llbracket EX\ EF\ p \rrbracket$ |
| ▪ $p\ AU\ q = q \vee (p \wedge AX\ (p\ AU\ q))$ | ▪ $\llbracket p\ AU\ q \rrbracket = \llbracket q \rrbracket \cup (\llbracket p \rrbracket \cap \llbracket AX\ (p\ AU\ q) \rrbracket)$ |
| ▪ $p\ EU\ q = q \vee (p \wedge EX\ (p\ EU\ q))$ | ▪ $\llbracket p\ EU\ q \rrbracket = \llbracket q \rrbracket \cup (\llbracket p \rrbracket \cap \llbracket EX\ (p\ EU\ q) \rrbracket)$ |

11.3.3. Operadores CTL como puntos fijos

Dado un modelo $M = \langle (S, R), L \rangle$, definimos las funciones $\cdot : 2^S \rightarrow 2^S$ que dado un conjunto del conjunto de partes de S , nos devuelve los estados de ese conjunto que satisfacen la fórmula.

Notación usada en la clase:

$$\underbrace{AF\ p}_{(1)} = \underbrace{\lambda y.}_{(2)} \underbrace{p}_{(3)} \underbrace{\vee}_{(4)} \underbrace{AX\ y}_{(5)}$$

- (1) Nombre de la función, dada una fórmula p , vamos a crear una para cada operador CTL.
- (2) El parámetro de la función y es un conjunto de estados.
- (3) Es el subconjunto de estados de y que satisfacen p .
- (4) Es una operación sobre conjuntos: \vee es la unión y \wedge es la intersección.
- (5) Son los estados de S que cumplen $AX\ y$ ó $EX\ y$, osea que:
 - $AX\ y$ son todos los estados de S tal que todas sus transiciones van a algún estado de y .
 - $EX\ y$ son todos los estados de S tal que alguna de sus transiciones va a algún estado de y .

Funciones para punto fijo:

Punto fijo mínimo

- $\text{AF } p : \lambda y. p \vee \text{AX } y$
- $\text{EF } p : \lambda y. p \vee \text{EX } y$
- $p \text{ AU } q : \lambda y. q \vee (p \wedge \text{AX } y)$
- $p \text{ EU } q : \lambda y. q \vee (p \wedge \text{EX } y)$

Punto fijo máximo

- $p \text{ AG } p : \lambda y. p \wedge \text{AX } y$
- $p \text{ AG } p : \lambda y. p \wedge \text{AX } y$

Siguiendo la idea de los teoremas de más arriba, para conseguir un punto fijo mínimo, debemos aplicar, iterativamente, la función al conjunto vacío hasta que se estabilice la solución.

```

 $x \leftarrow \text{False}$ 
while  $x \neq f(x)$  do
   $x \leftarrow f(x)$ 
end while

```

Para los puntos fijos máximos, hacemos exactamente lo mismo pero, en vez de partir desde el conjunto vacío, partimos desde S (el conjunto de todos los estados).